Machine Learning Engineer Nanodegree

Capstone Project

David S. Price

June 5, 2019 – Updated June 14, 2019

I. Definition

Project Overview

My project proposal was inspired by a Kaggle competition project called "Malaria Cell Images Dataset - Cell Images for Detecting Malaria". The intent is to save humans by developing an algorithm to determine whether image cells show infestation by malaria. [1]

Malaria is "*a mosquito-borne disease caused by a parasite. People with malaria often experience fever, chills, and flu-like illness. Left untreated, they may develop severe complications and die. In 2016 an estimated 216 million cases of malaria occurred worldwide and 445,000 people died, mostly children in the African Region. About 1,700 cases of malaria are diagnosed in the United States each year. The vast majority of cases in the United States are in travelers and immigrants returning from countries where malaria transmission occurs, many from sub-Saharan Africa and South Asia.*"[2]

This project uses the dataset of images (which actually comes from the National Institutes of Health[3]) containing cells in two categories – those parasitized by malaria and those uninfected – and then developed a convolutional neural network to detect the differences with 95.03% accuracy compared to a benchmark of 53.88%. All files are available on my Github site other than the full set of images which is prohibitively large[8]:

https://github.com/davidsprice/Malaria-Detector-CNN

Problem Statement

The problem statement defined from the beginning was to design a multi-layer, fully connected convolutional neural network (CNN) of my own design which could be trained to detect with at least 80% accuracy those images of cells which have been parasitized by malaria from images of cells which have been uninfected (i.e. a classification problem) in the Kaggle Malaria imageset.[4]

To support this primary goal, a benchmark also had to be established to determine the new model's effectiveness and usefulness. With advice from my Udacity mentor, it was recommended a simple single-layer CNN would need to be developed to serve as the benchmark for the project.

A secondary goal was also established which was the desire to make the algorithm more accessible to the masses by being able to comfortably run in a CPU environment instead of being forced to use a GPU environment as not everyone has access to a GPU environment. While the extremely large dataset of over 27k images made this look like a daunting task, creating a careful balance of the batch size, number of epochs, number of steps per epoch, the number of validation steps per epoch, and the model size itself was viewed as the most likely way to achieve this secondary goal while still achieving the primary goal.

During the proposal phase, it was thought that transfer learning may help reduce the training and validation processing time. First, let's review what transfer learning is. "In Transfer Learning, the knowledge of an already trained Machine Learning model is applied to a different but related problem. For example, if you trained a simple classifier to predict whether an image contains a backpack, you could use the knowledge that the model gained during its training to recognize other objects like sunglasses."[5]

Basically, the intent is to exploit what has already been learned in one task and improve the generalization of another task, transferring the weights from Task A to a new Task B in a network.[5]

When reviewing the available project databases available on the web, similar databases that could be leveraged weren't readily available, thus the full extent of transfer learning was not explored as part of this project.

Metrics

The two primary metrics utilized as part of this project are:

1. Test Accuracy
2. Area Under the Curve (AUC) Receiver Operating Characteristics (ROC) curves

Below are more details on both.

1. Test Accuracy - The first one is test accuracy using the test set in the Keras evaluate_generator method which evaluates the model on a data generator. This will be performed for the benchmark model and the full model with inputs as shown below.

    evaluate_generator(generator = test_set, steps = NUMBER_TEST_SET_SAMPLES)

    where:

    test_set is the data set containing the test images

    NUMBER_TEST_SET_SAMPLES is the number of images in the test set

2. Area Under the Curve (AUC) Receiver Operating Characteristics (ROC) curves - An AUC - ROC curve is "a performance measurement for classification problem at various thresholds settings. ROC is a probability curve and AUC represents the degree or measure of separability. It tells how much model is capable of distinguishing between classes. Higher the AUC, better the model is at predicting 0s as 0s and 1s as 1s. By analogy, Higher the AUC, better the model is at distinguishing between patients with disease and no disease." [6] This makes use of ROC curves

extremely useful for classification problems as one can visually see how well the model performs at separating the two classes[6], in this case, parasitized vs. uninfected cells.

Let's start with some terminology.

TPR (True Positive Rate/Recall/Sensitivity)
TPR = TP/(TP + FN)

Specificity
Specificity = TN/(TN + FP)

FPR (False Positive Rate)
FPR = 1 – Specificity

Where:
TP = True Positives
TN = True Negatives
FP = False Positives
FN = False Negatives

An AUC-ROC curve is produced by plotting the true positive rate (TPR), aka recall or sensitivity, on the y-axis and the false positive rate (FPR), which is 1 minus the specificity, along the x-axis. A model which is good at separating and classifying will have an AUC near 1.

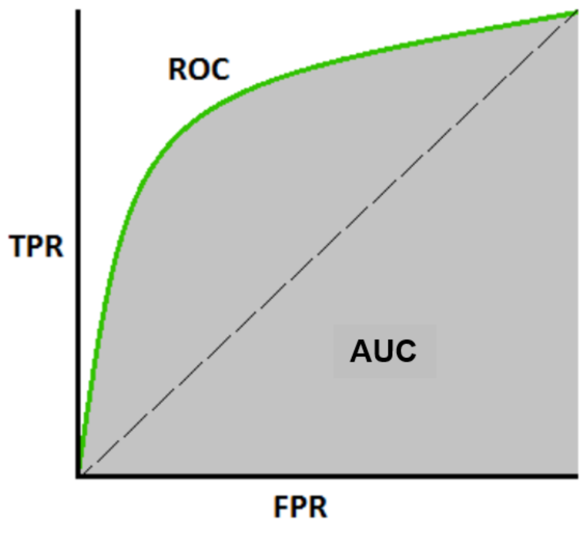


Figure 1 - Example AUC-ROC Curve

II. Analysis

Data Exploration

The dataset used was the set of images from the Kaggle project description[1], which in turn came from the National Library of Medicine[3]. In reviewing the data, the following observations were made:

1) The images will be subdivided into three categories – a training set, a validation set, and a test set.
2) The database contains a total of 13,779 parasitized images and 13,779 uninfected images.
3) These will each be split into a training set (60%), a validation set (20%), and a test set (20%).
4) The images are of different sizes ranging from ~133 x ~133 pixels in width and height, so they will be re-sized to a standard square shape of 64 pixels.
5) The images are smaller than used during a similar exercise during the course-work (e.g. the skin cancer mini-project), so computational processing time is not considered to be an issue.
6) The dataset is not considered to be imbalanced.
7) No abnormalities or other characteristics are present such as missing values.

Statistics were produced on the image as shown in the Table 1 below. This proved to be very enlightening data as it shows the extreme range of the smallest and largest width and height pixel values. While selecting a square shape of 64 x 64 pixels produced a high accuracy, key image data may be deleted from images with larger width or height sizes. The average mean pixel intensities of the image datasets are fairly close, ranging from 109.1 to 121.9 on the RGB scale.

Table 1 - Image Dataset Statistics

	Training Dataset		Valid Dataset		Test Dataset	
Characteristic	Parasitized	Uninfected	Parasitized	Uninfected	Parasitized	Uninfected
# of files	9645	9645	1378	1378	2756	2756
Average Width (pixels)	133.1	131.2	132.4	131.5	136.1	131.8
Average Height (pixels)	134.5	131.7	132.1	130.6	135.2	131.7
Average Mean Pixel Intensities (RGB)	114.5	121.9	112.1	115.9	109.1	115.8
Smallest Width (pixels)	61	58	46	61	55	49
Largest Width (pixels)	394	247	241	217	265	226
Smallest Height (pixels)	46	55	52	61	40	49
Largest Height (pixels)	385	235	325	214	346	229

Exploratory Visualization

The primary concern of the data is the varying size of the images in terms of height and width. These parameters for the images range from averages of ~133 x ~133 pixels in width and height, so reducing the size to a standard square shape of 64 x 64 pixels may cause some relevant parts of the images, which would otherwise show a cell to be parasitized, to be cut, thus leading to potentially false negatives.

Sample cell images post-processing are shown in Figure 2 below, where [1,0] means parasitized and [0,1] means uninfected. The characteristic dots indicating infection are still visible and nearer the center of the cell image, thus it is expected that re-shaping the images to a 64 x 64 pixel size should produce satisfactory results.

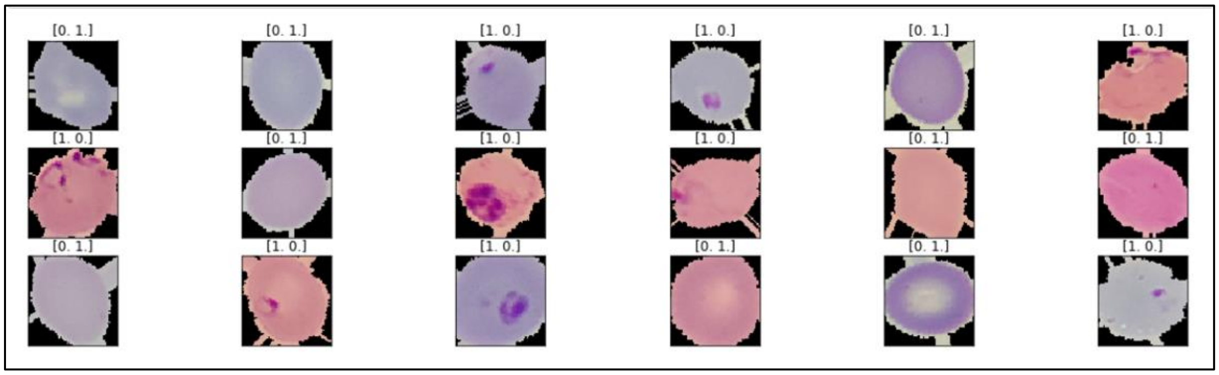


Figure 2 - Sample Cell Images

After developing the model and producing results, consideration may have to be given to what size to shape the image used in the algorithm. As the results will show, however, using a 64 x 64 pixel size produced very high accuracy.

Algorithms and Techniques

The solution to the project will be a convolutional neural network (CNN) which can detect with at least 80% accuracy images of cells which have been parasitized by malaria from images of cells which have been uninfected.

A convolutional network is “a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.”[9]

Figure 3 depicts an example 2-layer CNN where the input image is analyzed through two convolutional layers which is referred to as Feature Learning. The resultant is then flattened and then fully connected to an output classification layer, referred to as Classification.

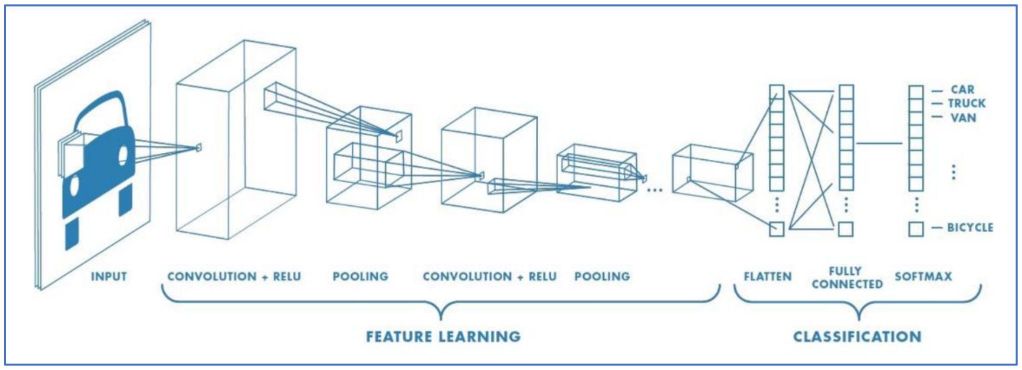


Figure 3 - Example 2-layer CNN[9]

Figure 4 depicts how the convolution works. The image matrix is "convolved" or crossed with another matrix, a filter, with a smaller resultant output matrix. In basic terms, the operation is "an element-wise multiplication between the two matrices and finally an addition of the multiplication outputs. The final integer of this computation forms a single element of the output matrix."[10] In this example, the value of 1 highlights brightness while 0 highlights gray and -1 highlights the darkness from the original image when the filter slides over top of the image matrix.[10]

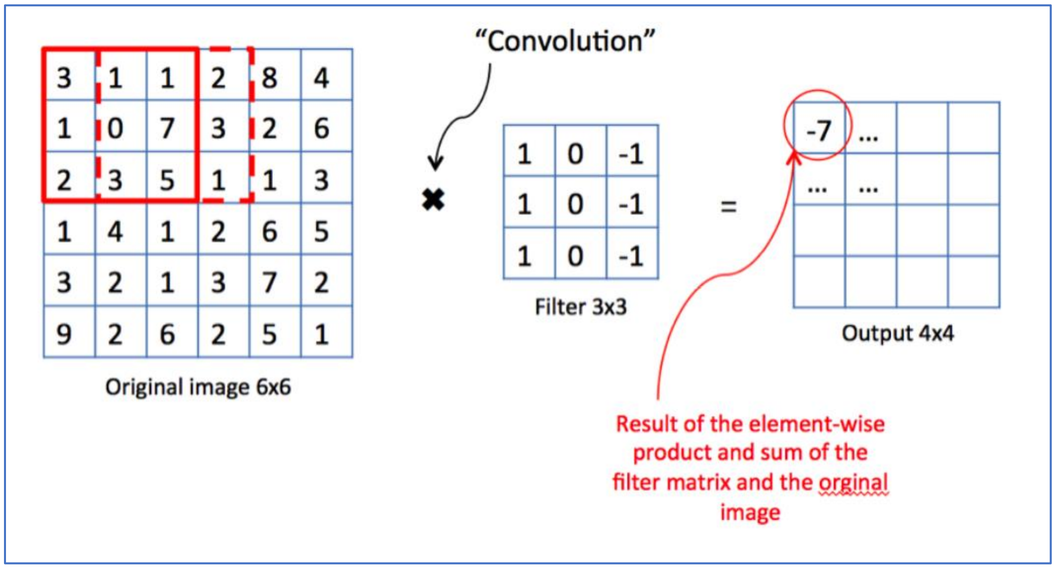


Figure 4 - Example Convolution

Figure 5 depicts how a filtered subset of the image matrix is transformed into a "pooling" layer by taking the largest element from each region.[10]

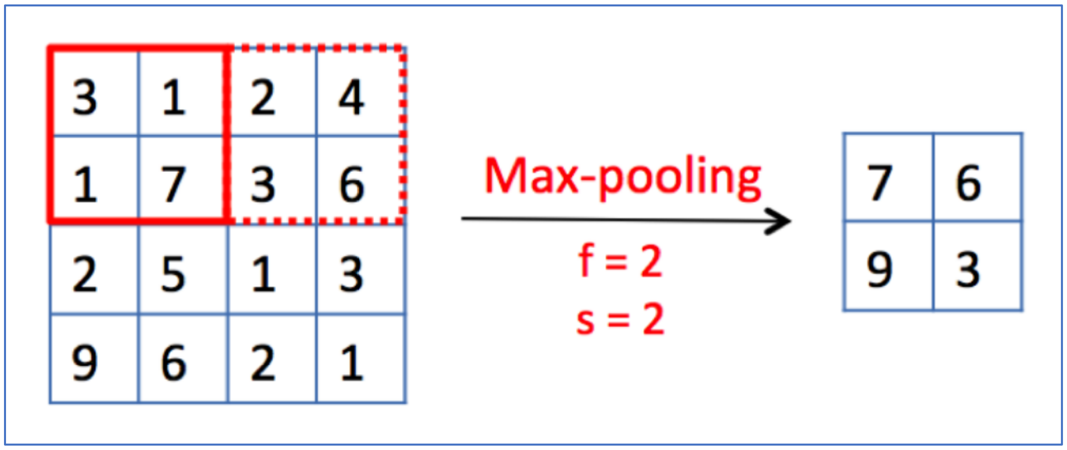


Figure 5 - Example of Max Pooling

Figure 6 depicts an example fully connected layer of a CNN. Starting with unrolling or flattening the matrix into a single 1D layer matrix, the fully connected layer behaves like a "standard" single neural network layer with a weight matrix W and bias b, and ending with a probability ranging from 0 to 1 for each of the classes of the data.

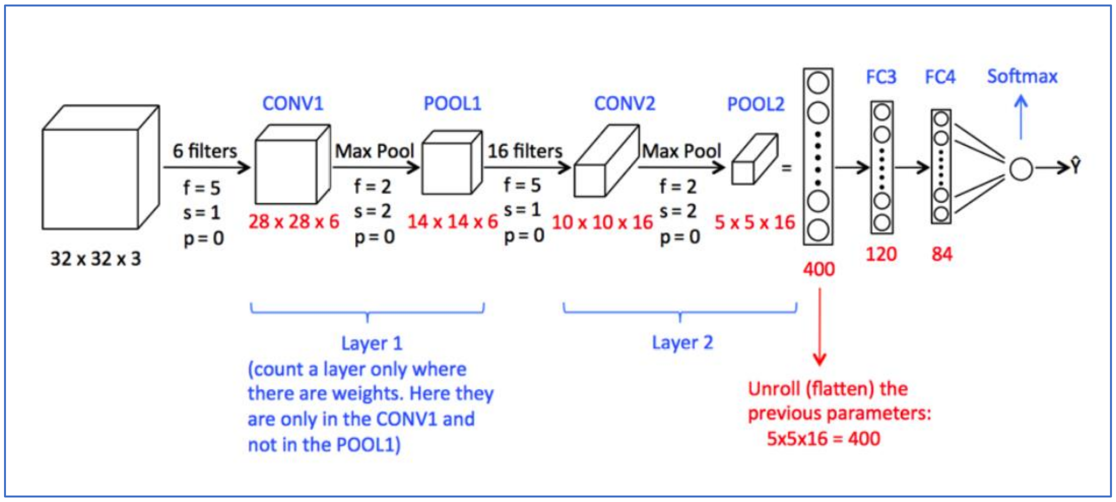


Figure 6 - Example CNN with Fully Connected Layer

A common feature that is often used with neural networks is a term called “dropout”. Dropout is “a regularization method that approximates training a large number of neural networks with different architectures in parallel. During training, some number of layer outputs are randomly ignored or “dropped out.” This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different “view” of the configured layer. By dropping a unit out, we mean temporarily removing it from the network, along with all its incoming and outgoing connections” [11]

While a human can look at an image of a dog and see a dog, a computer sees just three levels of matrices of RGB (Red-Green-Blue) numbers. A CNN is able to “successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.”[9]

The intent is to use a 3-layer, fully connected CNN algorithm with 2 “dropouts” to train the model and then validate. The first layer of the network tries to learn patterns and edges, the second layer tries to understand the shape/color and other stuff, and the final layer called Feature layer/Fully Connected Layer tries to classify the image.[13] Once done, then the model will be used with the test image set to produce the same metrics as identified in Section I:

1) Test accuracy using the test set in the evaluate generator code which evaluates the model on a data generator.
2) Area Under the Curve (AUC) Receiver Operating Characteristics (ROC) curve

These results will then be compared against the benchmark metrics. A confusion matrix will also be produced to help visualize the output.

An important consideration in terms of the CNN architecture is the optimizer of choice. For this project, the Adam optimizer algorithm was selected, which is an optimization algorithm to update network weights iterative based in training data that can used in place of the classical stochastic gradient descent procedure.[12] Benefits of using Adam on optimization problems, include the following:

- Straightforward to implement.
- Computationally efficient.
- Little memory requirements.
- Invariant to diagonal rescale of the gradients.
- Well suited for problems that are large in terms of data and/or parameters.
- Appropriate for non-stationary objectives.
- Appropriate for problems with very noisy/or sparse gradients.
- Hyper-parameters have intuitive interpretation and typically require little tuning.[12]

Benchmark

There is no pre-existing benchmark for this project other than a comment on the Kaggle website indicating another contributor was able to achieve 97% accuracy[4], therefore, one of the first steps in the project was to establish a baseline score for the dataset using a simple one-layer CNN, and producing the same metrics as identified in Section I:

1) Test accuracy using the test set in the evaluate generator code which evaluates the model on a data generator.
2) Area Under the Curve (AUC) Receiver Operating Characteristics (ROC) curve

Once the benchmark and full-up model metrics are produced, they will be compared and analyzed for differences. A confusion matrix will also be produced to help visualize the output.

III. Methodology

Overview of Top-Level View of Processing

The code is structured in a Jupyter notebook as follows:

- Import needed libraries and packages
- Establish Control Parameters for the various output options available
- Pre-Process Cell Images
- Set training parameters
- Data Visualization
- Create and Run CNN Model (Benchmark and Full)
  - Create and compile the benchmark model (a one-layer CNN) and the full model (a three-layer fully-connected CNN)
  - Train each model and save best weights
  - Load the model weights with the best validation loss
  - Run the predict_generator method to create each model's prediction file (for use later to produce the ROC and CM)
  - Calculate the Classification Accuracy of each model on the Test Set

- Evaluate model's performance:
    - Receiver Operating Characteristic Curve (ROC)
    - Confusion Matrix (CM)

The code is structured such that it expects the following subfolders to exist:

- /cell_images_full - for storing the complete set of images (all 27k images) - more details below
- /cell_images_subset - for storing a subset of images (200 images) - more details below
- /prediction_files - for storing the prediction from the models using the test set of images
- /ground_truth_files - for storing the "truth" of image classification (parasitized or uninfected)
- /saved_models - for storing the best weights from each model

Data Preprocessing

Data Preprocessing consisted of the following steps:

1) Import needed libraries and packages
2) Establish Control Parameters for the various output options available
3) Pre-Process Cell Images
4) Set training parameters
5) Data Visualization

Here are the data preprocessing steps in more detail:

1) Import needed libraries and packages

```
#Import the Keras libraries and packages
import keras
from keras.callbacks import ModelCheckpoint
from keras.layers import Conv2D
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.layers import GlobalAveragePooling2D
from keras.layers import MaxPooling2D
from keras.models import Sequential
from keras.preprocessing.image import ImageDataGenerator

#import others
import itertools
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import sys
from sklearn.datasets import load_files
from sklearn.metrics import roc_curve, auc, confusion_matrix
import sys
```

2) Establish Control Parameters for the various output options available

Given the enormous size of the image database, it's important to know in advance what the user intends to accomplish with running the code in this notebook as forgetting a parameter can lead to hours of processing which may not be desired at that time.

The user may set the following parameters in this section:

- USE_FULL_IMAGE_SET
  - True = evaluate full image set (>27k files)
  - False = evaluate just a subset image set (200 files)
- RUN_BENCHMARK_CODE
  - True = run the "benchmark" code in the notebook to establish the 1-layer CNN benchmark performance
  - False = do not run the benchmark code

3) Pre-Process Cell Images

Set Image File Path - First, download the image set from "Malaria Cell Images Dataset - Cell Images for Detecting Malaria"[1], and then establish the following folder structure of the same folder in which the notebook is being run.

Structure for full image set (>27k images):

```
/cell_images_full
        /train
                /parasitized - 9645 images
                /uninfected - 9645 images
        /test
                /parasitized - 2756 images
                /uninfected - 2756 images
        /valid
                /parasitized - 1378 images
                /uninfected - 1378 images
```

Structure for subset image set (200 images) for rapid code development:

```
/cell_images_subset
        /train
                /parasitized - 70 images
                /uninfected - 70 images
        /test
                /parasitized - 20 images
                /uninfected - 20 images
        /valid
                /parasitized - 10 images
                /uninfected - 10 images
```

For a complete listing of how the images were divided between the train, validation, and test datasets for the results contained herein, refer to the following files, available on my Github site[8]:

- /ground_truth_files/ground_truth_full - COMPLETE.xls
- /ground_truth_files/ground_truth_subset - COMPLETE.xls

Create Tensor Image Data - Next, call ImageDataGenerator to generate batches of tensor image data with real-time data augmentation for the train, valid, and test sets.

```
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   rotation_range = 40,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)

test_datagen = ImageDataGenerator(rescale = 1./255)

validation_datagen = ImageDataGenerator(rescale = 1./255)
```

Create Tuples of Data for Each Image Data Set - Lastly, apply method "flow_from_directory" which takes the path to a directory & generates batches of augmented data, returning a "DirectoryIterator" yielding tuples of (x, y) where x is a numpy array containing a batch of images with shape (batch_size, target_size, channels) and y is a numpy array of corresponding labels. Do this to create the training_set and valid_set. Do the same for the test_set, except set the batch size to 1, which is critical to ensure the files align with the predictions section of code later on.

```
BATCH_SIZE = 32

training_set = train_datagen.flow_from_directory(training_set_image_path,
                                                 target_size = (64, 64),
                                                 batch_size = BATCH_SIZE,
                                                 class_mode = 'categorical')

validation_set = validation_datagen.flow_from_directory(validation_set_image_path,
                                                 target_size = (64, 64),
                                                 batch_size = BATCH_SIZE,
                                                 class_mode = 'categorical')

#Set batch_size to 1 to force 1 entry per image - critical for the prediction step later on
test_set = test_datagen.flow_from_directory(test_set_image_path, target_size = (64, 64),
                                                 batch_size = 1,
                                                 class_mode = 'categorical')
```

4) Set training parameters

First, set the following parameters based on the length (i.e. number of files) of the datasets:

```
NUMBER_TEST_SET_SAMPLES = length of the test dataset
NUMBER_VALIDATION_SET_SAMPLES = length of validation dataset
```

Then, set the following training parameters:

```
BATCH_SIZE = 10
NUMBER_STEPS_PER_EPOCH = NUMBER_TEST_SET_SAMPLES/BATCH_SIZE
NUMBER_OF_EPOCHS = 10
NUMBER_VALIDATION_STEPS = NUMBER_VALIDATION_SET_SAMPLES/BATCH_SIZE
```

5) Data Visualization

Seeing the data and fully understanding its limitations and potential is critical to ensuring successful algorithm development. To enable this understanding, a sample of 18 training images were displayed where *x* was the processed image and *y* was the classification or label.

```
#Plot the first 18 training images, including class or label where:
# [1,0] = parasitized
# [0,1] = uninfected
fig = plt.figure(figsize=(20,5))
for i in range(18):
    x,y = training_set.next()
    for j in range(0,1):
        image = x[j]
        ax = fig.add_subplot(3, 6, i + 1, xticks=[], yticks=[])
        ax.imshow(np.squeeze(x[j]))
        ax.set_title(str(y[j]))
plt.show()
```

Refer to Figure 2 of section **II. Analysis - Exploratory Visualization** for results.

Implementation

Once pre-processing steps are completed, the benchmark and full-model CNNs were then implemented as defined below.

Benchmark

If the RUN_BENCHMARK_CODE parameter is set, then go through the following steps to establish a benchmark performance:

1. Create and compile a one-layer CNN model with the # of filters set to 1
2. Train the benchmark CNN model and save best weights
3. Load the benchmark model weights with the best validation loss
4. Run the Keras predict_generator method to create the benchmark_prediction file (for use later to produce the ROC and CM)
5. Calculate the Classification Accuracy on the Test Set

Below are those steps in more detail:

1. Create and compile a one-layer CNN model

The benchmark model design is shown below:

```
#Initialize the CNN
benchmark_model = Sequential()

#Add convolution layer
benchmark_model.add(Conv2D(1, (3, 3), input_shape = (64, 64, 3), activation = 'relu'))

#Add pooling layer
benchmark_model.add(MaxPooling2D(pool_size = (2, 2)))

#Add flattening layer
benchmark_model.add(Flatten())

#Add fully connected layer
benchmark_model.add(Dense(units = 64, activation = 'relu'))
benchmark_model.add(Dense(units = 2, activation = 'softmax'))

#Compile the CNN
benchmark_model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics =
['accuracy'])
```

2. Train the benchmark CNN model and save best weights

Below is the implementation to train the benchmark CNN model and save the best weights.

```
# train the benchmark model if mode activated
if RUN_BENCHMARK_CODE == True:
   benchmark_checkpointer = ModelCheckpoint(
                                             filepath='saved_models/benchmark_model.weights.best.hdf5',
                                                     verbose=1,
                                                     save_best_only=True)
```

```
#Fit the benchmark model
benchmark_model.fit_generator(training_set,
                              steps_per_epoch = NUMBER_STEPS_PER_EPOCH,
                              epochs = NUMBER_OF_EPOCHS,
                              callbacks=[benchmark_checkpointer],
                              validation_data = validation_set,
                               validation_steps = NUMBER_VALIDATION_STEPS)
```

3. Load the benchmark model weights with the best validation loss

Below is the implementation to load the benchmark model weights with the best validation loss.

```
if RUN_BENCHMARK_CODE == True:
    benchmark_model.load_weights('saved_models/benchmark_model.weights.best.hdf5')
```

4. Run the Keras predict_generator method to create the benchmark_prediction file (for use later to produce the ROC and CM)

Below is the implementation to create the predictions based on the benchmark model.

```
#Run prediction generator (for use with ROC and CM later) if mode activated
if RUN_BENCHMARK_CODE == True:

    benchmark_predict = benchmark_model.predict_generator(test_set,
                                                          steps = NUMBER_TEST_SET_SAMPLES)
```

5. Calculate the Classification Accuracy on the Test Set

Below is the implementation for computing the classification accuracy of the benchmark model using the test set.

```
if RUN_BENCHMARK_CODE == True:
    benchmark_score = benchmark_model.evaluate_generator(
                                    test_set,
                                    steps = NUMBER_TEST_SET_SAMPLES)
    benchmark_accuracy = 100*benchmark_score[1]
    print('Benchmark Test accuracy: %.4f%%' % benchmark_accuracy)
```

Full-Model

The same sequence of steps is used to create the 3-layer fully connected CNN, also known as the full model.

1. Create and compile a three-layer fully-connected CNN model

2. Train the full CNN model and save best weights
3. Load the full model weights with the best validation loss
4. Run the Keras predict_generator method to create the full model's prediction file (for use later to produce the ROC and CM)
5. Calculate the Classification Accuracy on the Test Set

Below are those steps in more detail:

1. Create and compile a three-layer fully-connected CNN model

The 3-layer fully connected model design is shown below:

```
#Initialize the CNN
model = Sequential()

#Add 1st convolutional layer
model.add(Conv2D(32, (3, 3), input_shape = (64, 64, 3), activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2, 2)))

#Add 2nd convolutional layer
model.add(Conv2D(32, (3, 3), activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2, 2)))

#Add 3rd convolutional layer
model.add(Conv2D(32, (3, 3), activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2, 2)))

#Add fully connected layer
model.add(Dropout(0.3))
model.add(Flatten())
model.add(Dense(units = 128, activation = 'relu'))
model.add(Dropout(0.3))
model.add(Dense(units = 2, activation = 'softmax'))

#Compile the CNN
model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

2. Train the full CNN model and save best weights

Below is the implementation to train the full CNN model and save the best weights.

```
#Train the model
checkpointer = ModelCheckpoint(filepath='saved_models/model.weights.best.hdf5',
                               verbose=1,
                               save_best_only=True)
#Fit the model
model.fit_generator(training_set,
                    steps_per_epoch = NUMBER_STEPS_PER_EPOCH,
                    epochs = NUMBER_OF_EPOCHS,
                    callbacks=[checkpointer],
                    validation_data = validation_set,
                    validation_steps = NUMBER_VALIDATION_STEPS)
```

3. Load the full model weights with the best validation loss

Below is the implementation to load the full model weights with the best validation loss.

```
# Load the model weights with the best validation loss.
model.load_weights('saved_models/model.weights.best.hdf5')
```

4. Run the Keras predict_generator method to create the full model's prediction file (for use later to produce the ROC and CM)

Below is the implementation to create the predictions based on the full model.

```
predict = model.predict_generator(test_set,steps = NUMBER_TEST_SET_SAMPLES)
```

5. Calculate the Classification Accuracy on the Test Set

Below is the implementation for computing the classification accuracy of the full model using the test set.

```
score = model.evaluate_generator(test_set,
                                        steps = NUMBER_TEST_SET_SAMPLES)
accuracy = 100*score[1]
print('Test accuracy: %.4f%%' % accuracy)
```

Refinement

As expected with a project this large, many initial assumptions had to be tweaked or completely abandoned along the way to the final model solution. Here is a list of the prominent refinements:

1. Model Parameters - In the Kaggle project description, another Kaggle user claimed 97% accuracy with his model. To do this, the number of steps per epoch was set to 5000 and the number of validation steps set to 2000 when fitting the images to the CNN.[2] The processing time in a CPU-only environment was unacceptably slow, causing my PC to continually lock-up and the battery become noticeably hot, so presumably that model was run in a GPU environment. As a secondary objective of the project was to produce an algorithm that could be run in a CPU environment within a reasonable time limit, the parameters had to be adjusted as follows:
   Number of steps per epoch =

```
BATCH_SIZE = 10
NUMBER_STEPS_PER_EPOCH = NUMBER_TEST_SET_SAMPLES/BATCH_SIZE
NUMBER_OF_EPOCHS = 10
NUMBER_VALIDATION_STEPS =
```

NUMBER_VALIDATION_SET_SAMPLES/BATCH_SIZE

With:

NUMBER_TEST_SET_SAMPLES = 5512

NUMBER_VALIDATION_SET_SAMPLES = 2756

This produced the following:

NUMBER_STEPS_PER_EPOCH = 5512/10 = 551.2
NUMBER_VALIDATION_STEPS = 2756/10 = 275.6

The above combination produced very high accuracy while allowing the code to be portable to another CPU or even a GPU environment. The length of time to run the Keras fit_generator method went from locking up my computer (i.e. not running) to a run-time of about 75 minutes to go through 10 epochs.

2. Prediction File comparison against Truth – It took considerable time and research to determine that an error message rising from the code trying to compare the predictions file against the “truth” file was the result of the batch size used in creating the test dataset resulted in a mismatch in the ordering of files in the predictions file vs. that of the “truth” file. To resolve this, the batch size for the test set was simply set to a size of 1, so every file was loaded individually in the same order as the “truth” file.

Other than 2 above, no other complications occurred during the coding process.

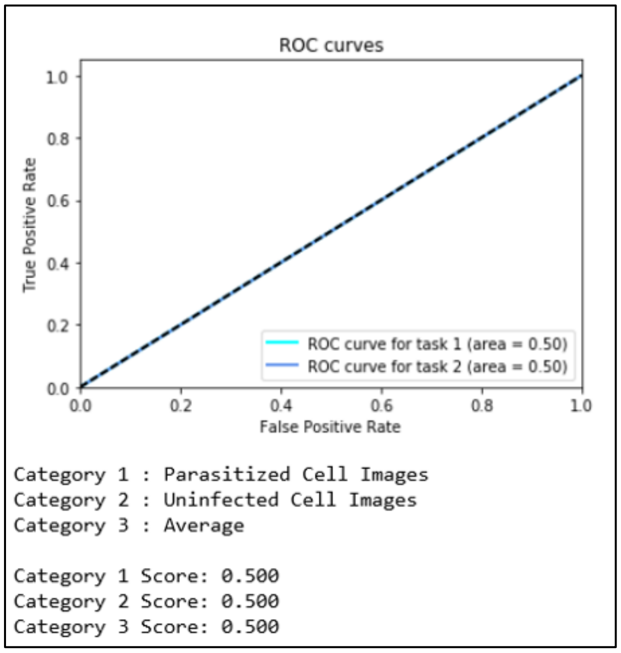
IV. Results

Model Evaluation and Validation

The accuracy of the test data using the Keras evaluate_generator method was as follows:

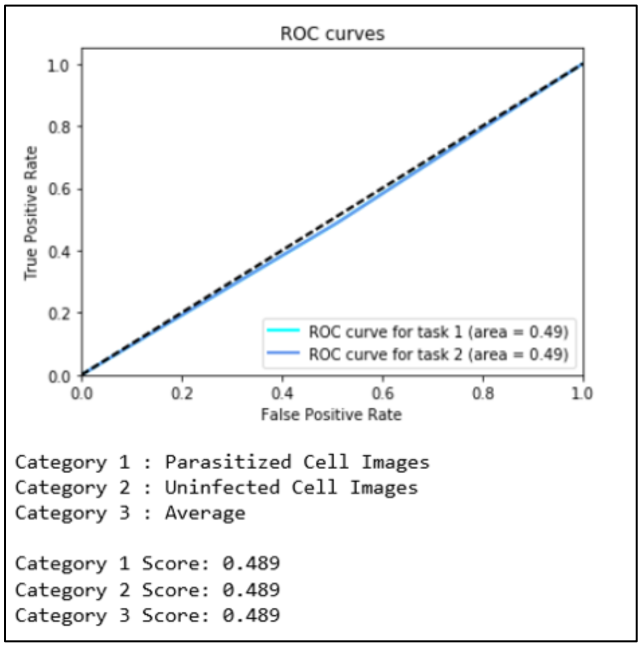
• Benchmark Model (one-layer CNN):	53.88%
• Full Model (3-layer, fully connected CNN):	95.03%

The resultant Area Under the Curve Receiver Operating Characteristics (AUC-ROC) curves for the benchmark model and the full model are shown in Figures 7 and 8, respectively.



```
Category 1 : Parasitized Cell Images
Category 2 : Uninfected Cell Images
Category 3 : Average

Category 1 Score: 0.500
Category 2 Score: 0.500
Category 3 Score: 0.500
```

Figure 7 - AUC-ROC results for Benchmark Model



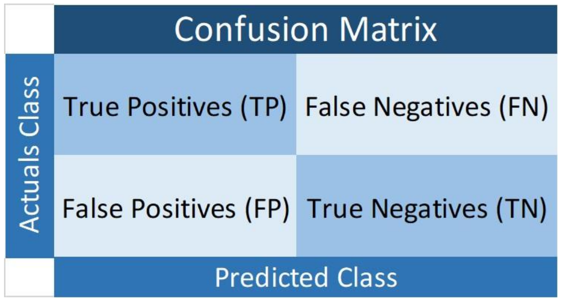
```
Category 1 : Parasitized Cell Images
Category 2 : Uninfected Cell Images
Category 3 : Average

Category 1 Score: 0.489
Category 2 Score: 0.489
Category 3 Score: 0.489
```

Figure 8 - AUC-ROC results for Full Model

The AUC-ROC curve for the benchmark model as shown in Figure 7 has an Area Under the Curve approximately equal to 0.5. This means the benchmark model has no discrimination capacity to distinguish between the two classes. The AUC-ROC curve for the full model as shown in Figure 8 is shows the same inability for the model to distinguish between the two classes. In reviewing the code and multiple runs of the output from the Keras predict_generator method, the predict_generator method is not a reliable means of predicting the classification of new images, presumably due to overfitting.

While not a metric per se, a confusion matrix (CM) was also produced. The CM is "a performance measurement for machine learning classification problem where output can be two or more classes. It is a table with 4 different combinations of predicted and actual values."[7] For this project, a CM table was produced for the true label vs. predicted label of parasitized vs. uninfected cell images to show true and false positive and negative predictions of the image classifer vs truth. Refer to Table 2 below.

Table 2 - Confusion Matrix



	Confusion Matrix	
Actuals Class	True Positives (TP)	False Negatives (FN)
	False Positives (FP)	True Negatives (TN)
	Predicted Class	

The resultant confusion matrix for the benchmark model and the full model are shown in Figures 9 and 10, respectively.

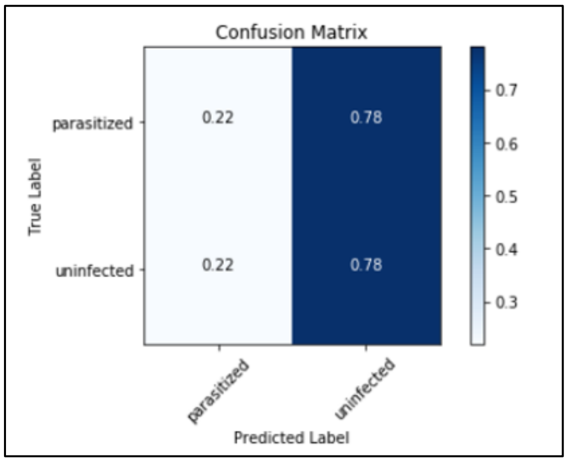


Figure 9 - Confusion Matrix Results for Benchmark Model

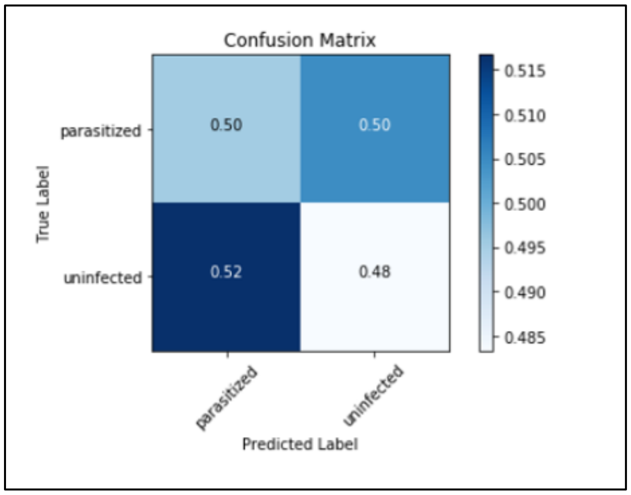


Figure 10 - Confusion Matrix Results for Full Model

For the benchmark model, the confusion matrix indicates a true positive rate of 22% and a false negative rate of 78%, and a false positive rate of 22% and a true negative rate of 78%. For the full model, the confusion matrix, however, indicates a true positive rate of 50% and a false negative rate of 50%, and a false positive rate of 52% and a true negative rate of 48%. In reviewing the code and multiple runs of the output from the Keras predict_generator method, the predict_generator method is not a reliable means of predicting the classification of new images, presumably due to overfitting.

Justification

The 3-layer, fully connected CNN proved it is much more accurate in predicting cell images that have been infected with malaria than the one-layer benchmark model, by a healthy margin of 41.15%. The accuracy of the test data using the Keras evaluate_generator method was as follows:

• Benchmark Model (one-layer CNN):	53.88%
• Full Model (3-layer, fully connected CNN):	95.03%
• Delta increase:	41.15%

V. Conclusion

Free-Form Visualization

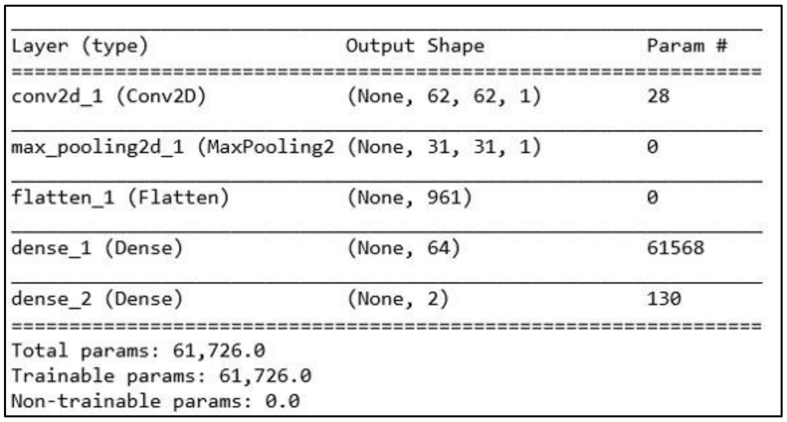
This project reinforced the potential power of AI techniques in helping solve real problems. As indicated in the intro, *445,000 people died, mostly children in the African Region, from malaria in 2016.*[2] *With an accuracy of 95.03% in the ability to detect the infection, potentially 422,883 people could have been saved worldwide. For the 1,700 cases of malaria in the United States each year*[2]*, that equates to 1,616 people that could have been saved or had the symptoms detected much earlier.*

Reflection

This was a very educational and enjoyable project. Not only did it allow me to work on a real-world issue, it helped reinforce the differences between working with a small dataset vs. a large dataset, and the tools needed (such as Keras' fit_generator method vs. just the fit method). The project really forced analysis of the data. When I first selected this project, it wasn't clear whether the image size could make a difference. While ultimately a square shape of 64 x 64 pixels was chosen, future experiments might be conducted to determine the impact of larger sizes.

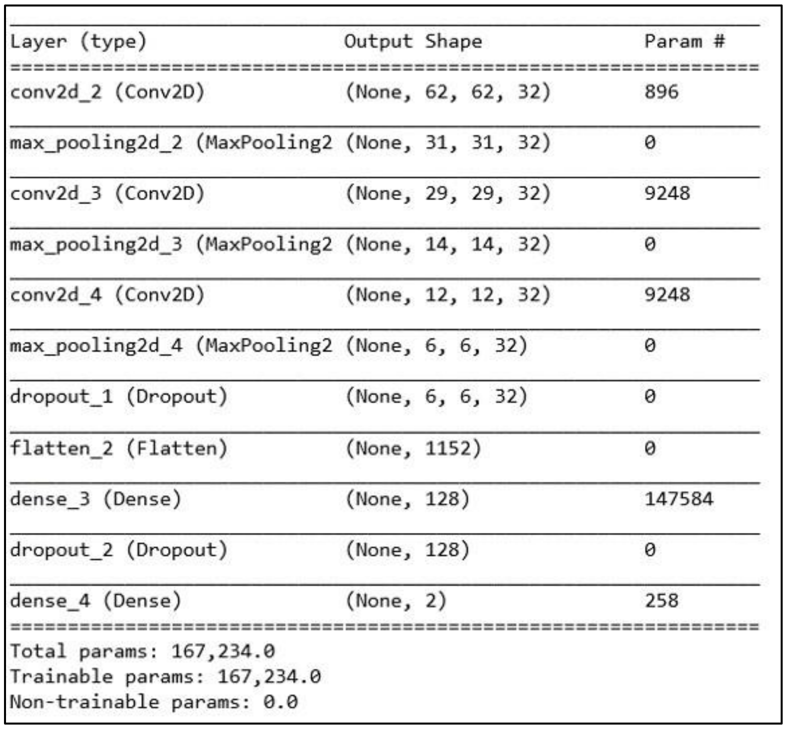
The first project submittal review comments also helped me understand the importance of the model parameters and how to calculate the trainable parameters.[13] This also helped discover a fundamental error in the original benchmark model. The intent was to create a very simple model to use for benchmarking. The original benchmark model had the number of filters set to 32 in the call to Conv2D function. This resulted in accuracies of upwards of 92-93%, certainly much higher than I would expect for a benchmark. Setting the number of filters to just 1, again to reflect a simple model, resulted in a more realistic starting point of just over 50%. This discovery process emphasized understanding the role of each model parameter.

The final model parameters of the benchmark model and the full model are shown in Figures 11 and 12, respectively. As can be seen, the number of trainable parameters in the benchmark model is much smaller than that of the full model (61,726 compared to 167,234). The benefit of having more trainable parameters is that many more complicated functions can be represented in a model than with fewer parameters. "The relationships that neural networks model are often very complicated ones and using a small network (adapting the size of the network to the size of the training set, i.e. making your data look big just by using a small model) can lead to the problem when your network is too simple and unable to represent the desired mapping (high bias)."[14] This appears to be the case with the benchmark model; it's a simple model but unable to represent the classification task needed.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 62, 62, 1)         28
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 31, 31, 1)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 961)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                61568
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 130
=================================================================
Total params: 61,726.0
Trainable params: 61,726.0
Non-trainable params: 0.0
```

Figure 11 - Model Parameters of Benchmark Model



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 62, 62, 32)        896
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 31, 31, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 29, 29, 32)        9248
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 12, 12, 32)        9248
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 6, 6, 32)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 6, 6, 32)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 1152)              0
_________________________________________________________________
dense_3 (Dense)              (None, 128)               147584
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_4 (Dense)              (None, 2)                 258
=================================================================
Total params: 167,234.0
Trainable params: 167,234.0
Non-trainable params: 0.0
```

Figure 12 - Model Parameters of Full Model

The project also reinforced the key process steps.

- Import needed libraries and packages
- Establish Control Parameters for the various output options available
- Pre-Process Cell Images
- Set training parameters
- Data Visualization
- Create and Run CNN Model (Benchmark and Full)
  - Create and compile the benchmark model (a one-layer CNN) and the full model (a three-layer fully-connected CNN)
  - Train each model and save best weights
  - Load the model weights with the best validation loss
  - Run the predict_generator method to create each model's prediction file (for use later to produce the ROC and CM)
  - Calculate the Classification Accuracy of each model on the Test Set
- Evaluate model's performance:
  - Receiver Operating Characteristic Curve (ROC)
  - Confusion Matrix (CM)

The final model does indeed meet my expectations and the objective of achieving at least 80% accuracy and the secondary objective of creating a model that is portable among CPU, or even GPU, environments.

Improvement

In terms of possible improvements, the image size is the easiest aspect to further explore. Would increasing the size from 64 x 64 pixels to, let's say, 128 x 128 pixels, improve results while still meeting the primary and secondary objectives? This is something I will explore on my own post-capstone project.

The other area of implementation that could possibly be improved is the prediction algorithm. While very pleased with the high accuracy of the 3-layer model, it's quite perplexing that the predict_generator resulted in very random classification of the cell images. Extensive manual analysis was extended to ensure the ROC and Confusion Matrix algorithms were displaying the correct computed values. Once determined they were displaying the computed values, it left me wondering what the underlying difference is between the evaluate_generator and the predict_generator methods. This would be a great area of additional research as the issue is most likely overfitting.

References

1 https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria

2 https://www.cdc.gov/parasites/malaria/index.html

3 https://ceb.nlm.nih.gov/repositories/malaria-datasets/

4 https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria/discussion/80214

5. https://towardsdatascience.com/transfer-learning-946518f95666

6. https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5

7. https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62

8. https://github.com/davidsprice/Malaria-Detector-CNN

9. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

10. https://medium.com/machine-learning-bites/deeplearning-series-convolutional-neural-networks-a9c2f2ee1524

11. https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/

12. https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

13. https://medium.com/@iamvarman/how-to-calculate-the-number-of-parameters-in-the-cnn-5bd55364d7ca

14. https://stats.stackexchange.com/questions/329861/what-happens-when-a-model-is-having-more-parameters-than-training-samples